WHAT IS CLAIMED IS:

1. A mutual exclusion mechanism comprising:

a biasable lock; and

acquisition and release sequences that, when executed by a thread to which bias has been directed, are free of atomic read-modify-write operations.

2. The mutual exclusion mechanism of claim 1,

wherein the acquisition and release sequences are further free of memory barrier operations, at least when executed by the thread to which bias has been directed.

3. The mutual exclusion mechanism of claim 1,

wherein the acquisition and release sequences include only read and write operations when executed by the thread to which bias has been directed.

4. The mutual exclusion mechanism of claim 1,

wherein the lock is initially unbiased.

5. The mutual exclusion mechanism of claim 1,

wherein the lock is biased on creation.

6. The mutual exclusion mechanism of claim 1,

wherein the lock is biased on acquisition.

7. The mutual exclusion mechanism of claim 1,

wherein the bias is directed to a thread other than a creating thread.

8. The mutual exclusion mechanism of claim 1,

wherein the acquisition sequence employs a programming construct that precludes reordering of a particular read before a particular write.

9. The mutual exclusion mechanism of claim 8,
wherein the precluded reordering includes reordering by a compiler.

10. The mutual exclusion mechanism of claim 8,
wherein the precluded reordering includes reordering upon execution by a processor.

11. The mutual exclusion mechanism of claim 8,
wherein the programming construct employs collocation of the target of the particular read and the target of the particular write.

12. The mutual exclusion mechanism of claim 8,
wherein the programming construct employs a memory barrier interposed between the particular read and the particular write.

13. The mutual exclusion mechanism of claim 12,
wherein the memory barrier corresponds to a membar operation.

14. The mutual exclusion mechanism of claim 12,
wherein the memory barrier results from a signal handler.

15. The mutual exclusion mechanism of claim 12,
wherein the memory barrier results from a cross-call.

16. The mutual exclusion mechanism of claim 8,
wherein the programming construct is membar-free.

17. The mutual exclusion mechanism of claim 8,
wherein the particular read loads lock status; and
wherein the particular write stores a quick lock indication.

18. The mutual exclusion mechanism of claim 8,
wherein the preclusion is based, at least in part, on characteristics of an implementation of a memory model.

19. The mutual exclusion mechanism of claim 8,

embodied in code compiled for execution on a processor that implements a total store order (TSO) memory model,

wherein the programming construct includes use of a store operation followed in program order by a load operation, the store and load operations having collocated targets that encode a quick lock indication and lock status, respectively.

20. The mutual exclusion mechanism of claim 1,

wherein the lock includes an MCS lock augmented to provide fast path acquisition and release sequences for the thread to which bias has been directed.

21. The mutual exclusion mechanism of claim 1,

wherein the lock includes an TATAS lock augmented to provide fast path acquisition and release sequences for the thread to which bias has been directed.

22. The mutual exclusion mechanism of claim 1,

wherein the lock includes a lock provided by a POSIX pthreads mutex library, augmented to provide fast path acquisition and release sequences for the thread to which bias has been directed.

23. The mutual exclusion mechanism of claim 1,

wherein the lock includes a monitor provided by a Java virtual machine implementation, the monitor augmented to provide fast path acquisition and release sequences for the thread to which bias has been directed.

24. The mutual exclusion mechanism of claim 1,

wherein the lock is rebiasable to another thread during course of a computation that employs the lock.

25. The mutual exclusion mechanism of claim 1,

wherein revocation of bias by, or on behalf of, a contending thread, is mediated, at least in part, using a signal handler.

26. The mutual exclusion mechanism of claim 1,
wherein revocation of bias by, or on behalf of, a contending thread, is mediated, at least in part, using a cross-call.

27. The mutual exclusion mechanism of claim 1,
wherein revocation of bias by, or on behalf of, a contending thread, is handled, at least in part, at a garbage collection safe point.

28. The mutual exclusion mechanism of claim 24, further comprising:
means for detecting a current level of contention; and
a rebiasing sequence that rebiases the lock in response to detection of an absence of contention.

29. The mutual exclusion mechanism of claim 28,
wherein the contention detection means accesses a request queue to identify the absence of contention.

30. The mutual exclusion mechanism of claim 1,
wherein the contention detection means employs an attempt counter to identify the absence of contention.

31. A biasable lock that provides at least two acquisition sequences, a fast path acquisition sequence for a thread to which the lock has been biased and a second acquisition sequence, the fast path acquisition sequence optimized with respect to the second acquisition sequence.

32. The biasable lock of claim 31,
wherein the fast path acquisition sequence is free of atomic read-modify-write operations.

33. The biasable lock of claim 31,

wherein the fast path acquisition sequence is free of memory barrier operations.

34. The biasable lock of claim 31,
wherein the second acquisition sequence implements one of an MCS lock, a TATAS lock, a lock consistent with that provided by a POSIX pthread Mutex library and a Java monitor.

35. The biasable lock of claim 31,
wherein the lock is further rebiasable.

36. A method of providing an efficient locking mechanism in program code, the method comprising:
instantiating a biasable lock; and
for a thread to which bias has been directed, releasing and acquiring the lock using fast path instruction sequences that are free of atomic read-modify-write operations.

37. The method of claim 36, further comprising:
directing the bias to the thread coincident with a first acquisition of the biasable lock.

38. The method of claim 36, further comprising:
directing the bias to the thread coincident with the instantiation.

39. The method of claim 36, further comprising:
directing the bias to the thread coincident with creation of an object.

40. The method of claim 36,
wherein directing of bias to the thread is performed by another thread.

41. The method of claim 36, further comprising:

for a thread other than the thread to which bias has been directed, acquiring the lock using an instruction sequence that unbiases the lock, if then biased.

42. The method of claim 36, further comprising:
rebiasing the lock to another thread.

43. The method of claim 36, further comprising:
executing the program code as a multi-threaded application,
the biasable lock allowing a single thread of the executing program code to repeatedly acquire and release the lock with extremely low overhead.

44. The method of claim 43,
after rebiasing to another thread, allowing the another thread to repeatedly acquire and release the lock with extremely low overhead.

45. The method of claim 36, further comprising:
executing the program code in a single-threaded execution environment,
wherein the program code is compiled with the biasable lock for execution on both the single-threaded execution environment and a multi-threaded execution environment, and
wherein the biasable lock allows the program code to run in the single-threaded execution environment without significant lock-related overhead.

46. A lock that maintains both a lock state and bias state, whereby acquisition of the lock by a thread to which bias has been directed is more efficient than acquisition of the lock by another thread.

47. The lock of claim 46,
wherein the lock is rebiasable to another thread during course of a computation that employs the lock.

48. The lock of claim 46, including acquisition and release sequences that, when executed by the thread to which bias has been directed, are free of atomic read-modify-write operations.

49. The lock of claim 48,

wherein the acquisition and release sequences are further free of memory barrier operations, at least when executed by the thread to which bias has been directed.

50. The lock of claim 48,

wherein the acquisition and release sequences include only read and write operations when executed by the thread to which bias has been directed.

51. The lock of claim 48,

wherein the acquisition sequence employs a programming construct that precludes reordering of a particular read before a particular write.

52. A computer program product including a mutual exclusion mechanism embodied therein, the computer program product embodied in a computer readable medium and comprising:

a data structure instantiable in memory of a processor to implement a lock that includes a bias attribute; and

a lock acquisition sequence of operations executable by the processor, the lock acquisition sequence having a fast path for a thread to which bias has been directed and a second path, the fast path optimized with respect to the second path.

53. The computer program product of claim 52,

wherein, when executed by the thread to which bias has been directed, the lock acquisition sequence is free of atomic read-modify-write operations.

54. The computer program product of claim 53,

wherein the acquisition sequence is further free of memory barrier operations, at least when executed by the thread to which bias has been directed.

55. The computer program product of claim 52, further comprising:
a lock release sequence of operations executable by the processor, the lock release sequence having a fast path for the thread to which bias has been directed and a second path, the fast path optimized with respect to the second path.

56. The computer program product of claim 52,
wherein the acquisition sequence employs a programming construct that precludes reordering of a particular read before a particular write.

57. The computer program product of claim 52,
wherein the lock is rebiasable to another thread during course of a computation that employs the lock.

58. A software method comprising:
biasing a lock to a first thread of execution; and
subsequently acquiring the lock for, or by, the first thread with computational overhead substantially less than for a second thread to which the lock is not currently biased.

59. The software method of claim 58,
wherein the lock acquiring, when performed by, or for, the first thread, is free of atomic read-modify-write operations.

60. The software method of claim 58,
wherein the lock acquiring, when performed by, or for, the first thread, is further free of memory barrier operations.

61. The method of claim 58, further comprising:
rebiasing the lock to a third thread of execution.

62. A computer program product encoding instructions that implement a biasable lock.

63. The computer program product of claim 62, further comprising:
a biasing sequence.

64. The computer program product of claim 62, further comprising:
a lock acquisition sequence that when executed by a thread to which bias has been directed, is free of atomic read-modify-write operations.

65. The computer program product of claim 62, further comprising:
a lock release sequence that when executed by a thread to which bias has been directed, is free of atomic read-modify-write operations.

66. The computer program product of claim 62, further comprising:
a rebiasing sequence.

67. The computer program product of claim 62,
embodied in at least one computer readable medium selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.

68. A computer program product encoding instructions that implement a biasable lock suitable for execution as either a single-threaded and a multi-threaded computation, wherein post-biasing, no atomic operations are executed in the acquisition or release, by a bias-holding thread, of the biasable lock.

69. The computer program product of claim 62,
wherein, when executed as the single-threaded computation, the biasable lock imposes minimal lock acquisition and release overhead.

70. The computer program product of claim 62,

wherein, when executed as the multi-threaded computation, the biasable lock imposes minimal lock acquisition and release overhead when repeatedly acquired and released by the bias-holding thread.

71. The computer program product of claim 62,

embodied in at least one computer readable medium selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.

72. A method of making a single computer program product suitable for efficient execution as both a single-threaded computation and a multi-threaded computation, the method comprising:

structuring a computation as a potentially multithread computation;

mediating at least some sources of contention in the multithreaded computation using a biasable locking mechanism, and introducing the instances of the biasable locking mechanism into program code;

compiling the program code; and

encoding the compiled program code, including the instances of the biasable locking mechanism, in a computer program product.

73. The method of claim 72,

wherein the encoding includes transferring the compiled program code onto at least one computer readable medium selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.